

State Machines

15-494 Cognitive Robotics
David S. Touretzky &
Ethan Tira-Thompson

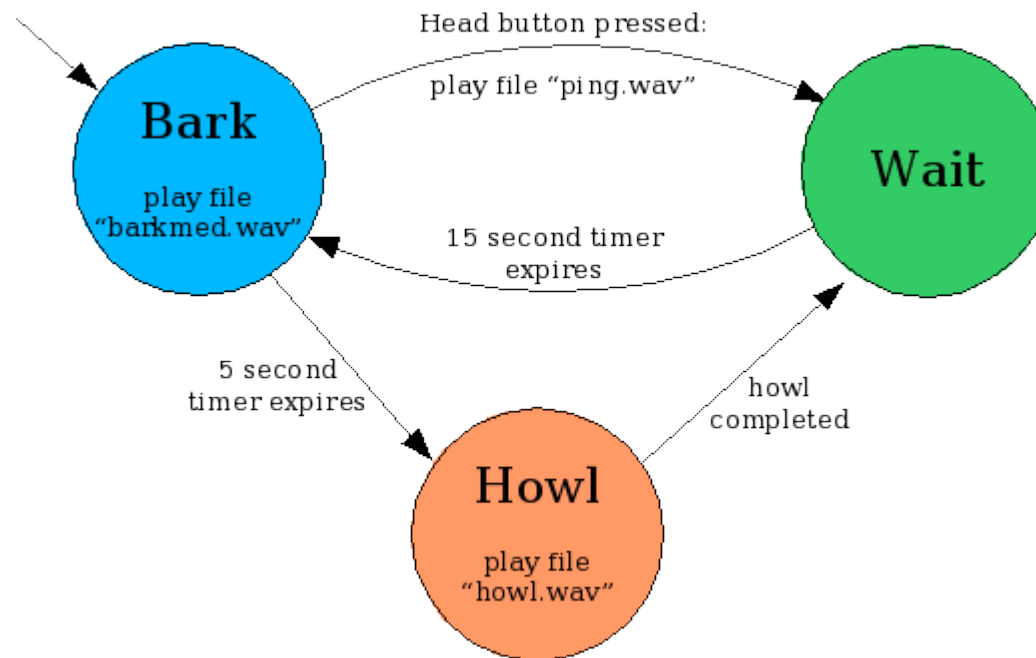
Carnegie Mellon
Spring 2009

Robot Control Architectures

- State machines are the simplest and most widely used robot control architecture.
- Easy to implement; easy to understand.
- Not very powerful:
 - Action sequences must be laid out in advance, as a series of state nodes.
 - No dynamic planning.
 - Failure handling must be programmed explicitly.
- But a good place to start.

Basic Idea

- Robot moves from state to state.
- Each state has an associated action: *speak*, *move*, etc.
- Transitions triggered by sensory events or timers.

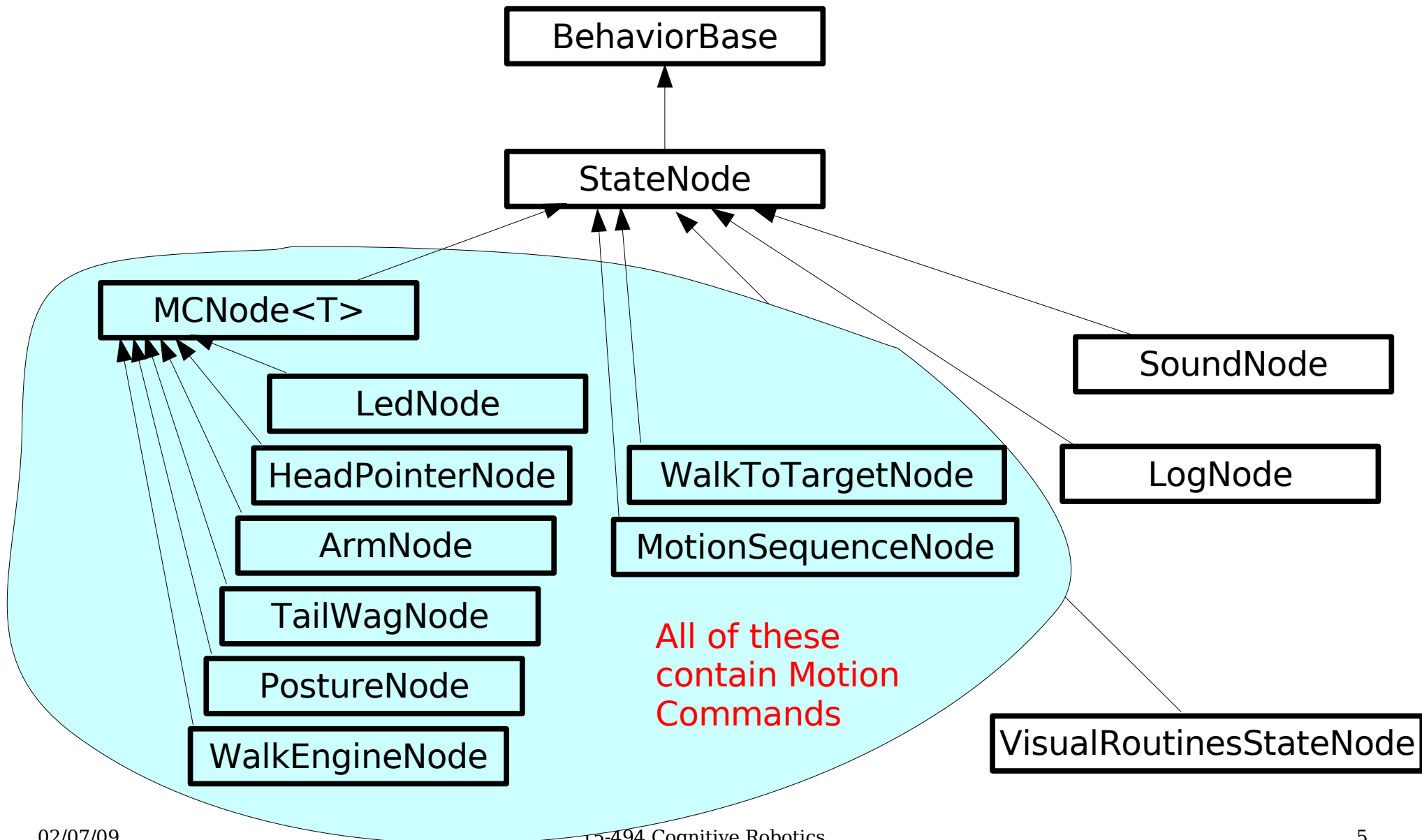


Tekkotsu State Nodes

- In Tekkotsu, state machine nodes are *behaviors*.
- StateNode is a child of BehaviorBase.
- To enter a state, call its start() method, which will call its DoStart() method if one has been supplied.
- To leave a state, call its stop() method.
- StateNodes can listen for and process events just like any other behavior.

Types of State Nodes

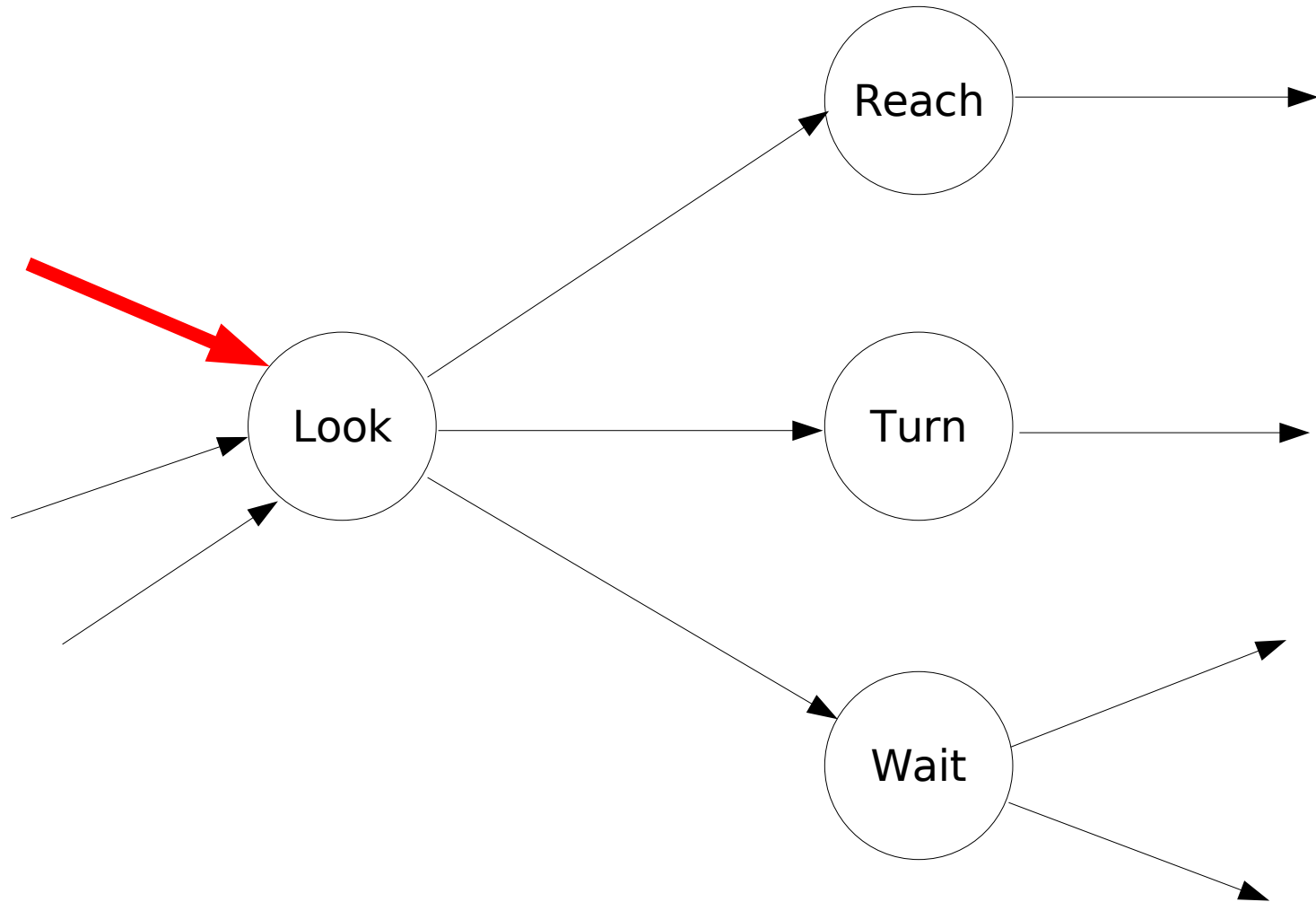
- State nodes encapsulate complex actions, such as creating and launching a motion command.



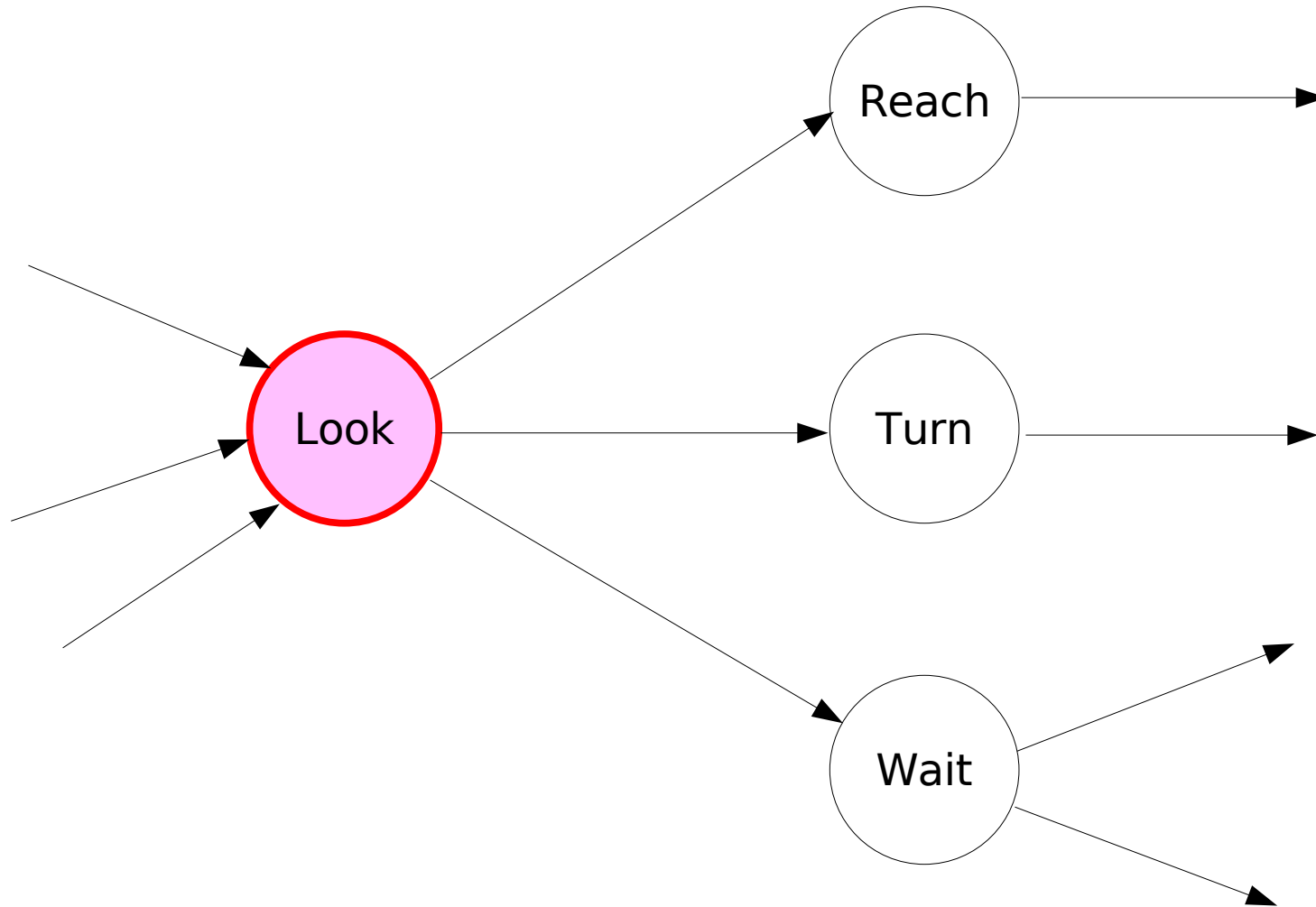
Transitions

- Transitions in Tekkotsu are also behaviors.
 - Transition and StateNode are *both* subclasses of BehaviorBase.
- A transition's start() is called whenever its source state node becomes active.
- Transitions listen for sensor, timer, or other events, and when their conditions are met, they *fire*.
- When a transition fires, it deactivates its source node(s) and then activates its destination node(s).

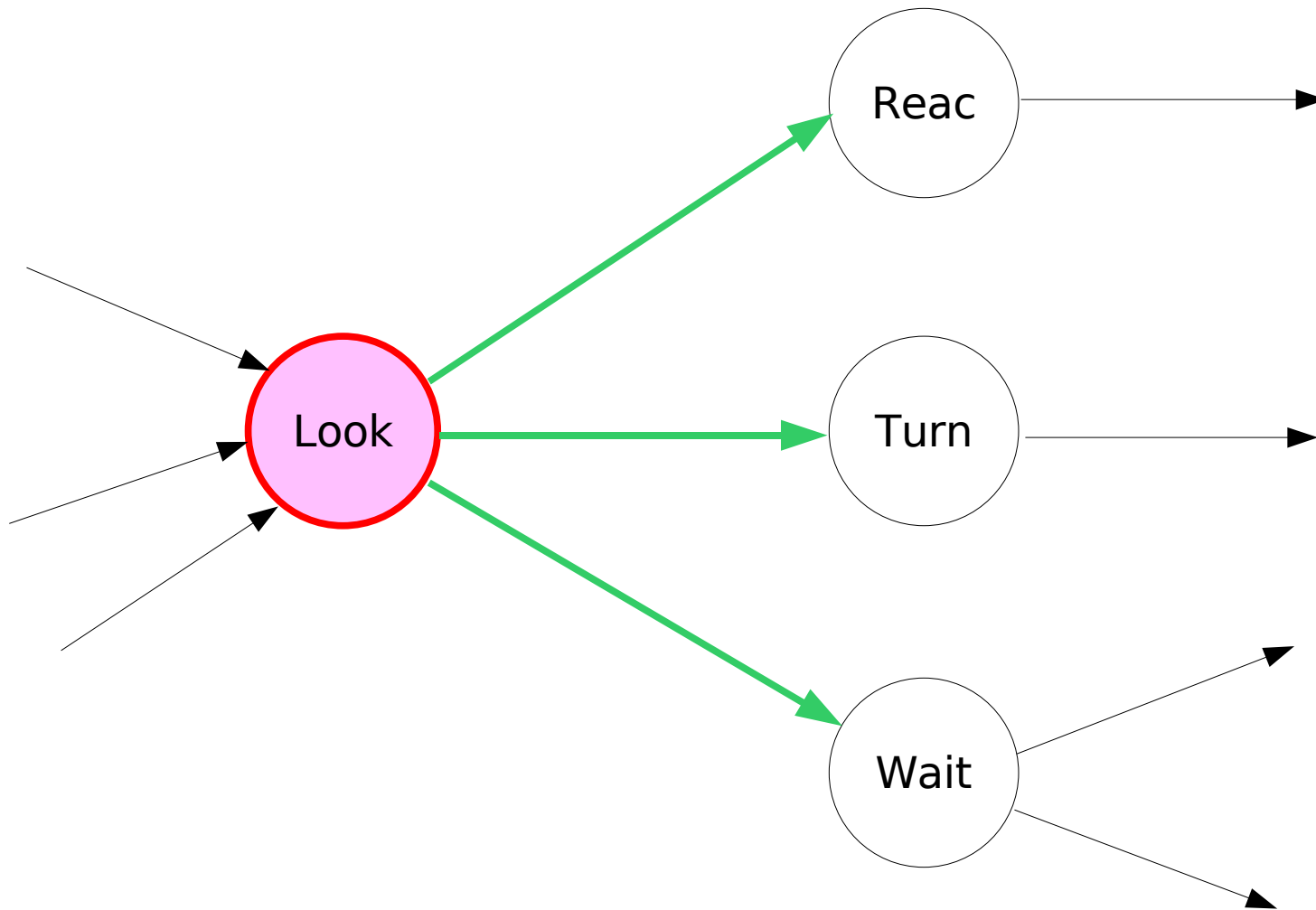
Transition firing activates state node Look.



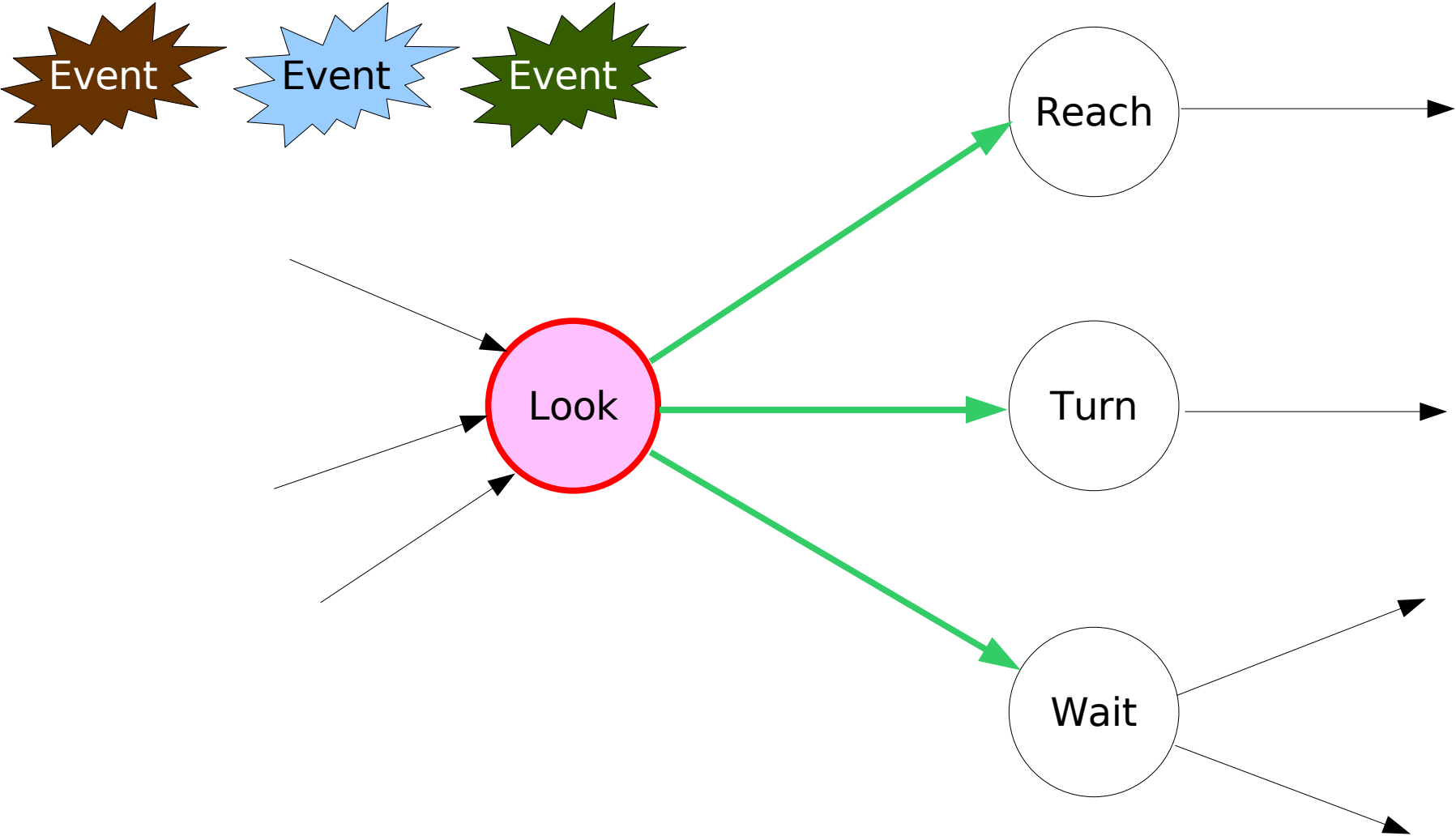
Look's start() calls StateNode::start().



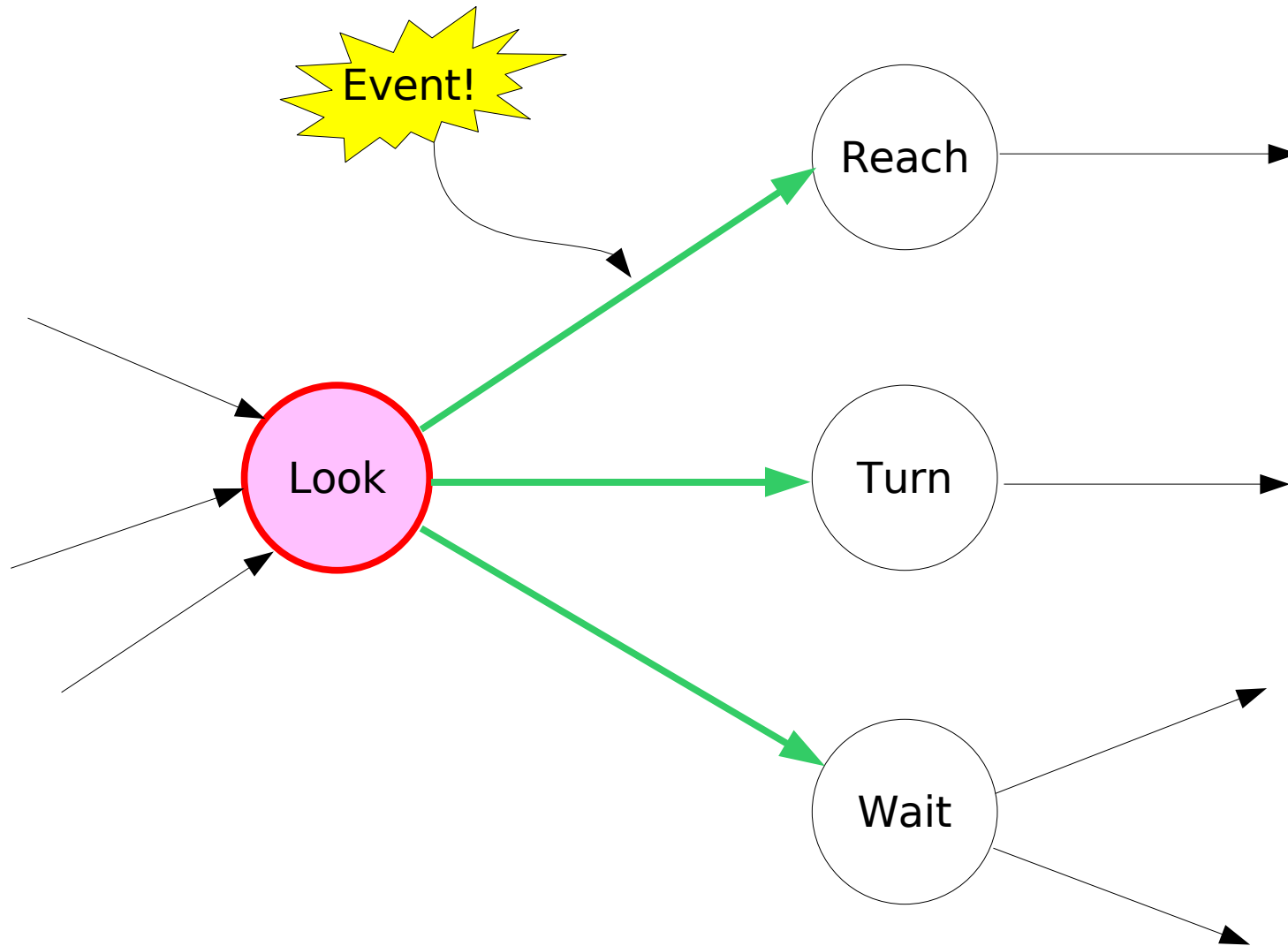
Outgoing transitions become active and begin listening for events.



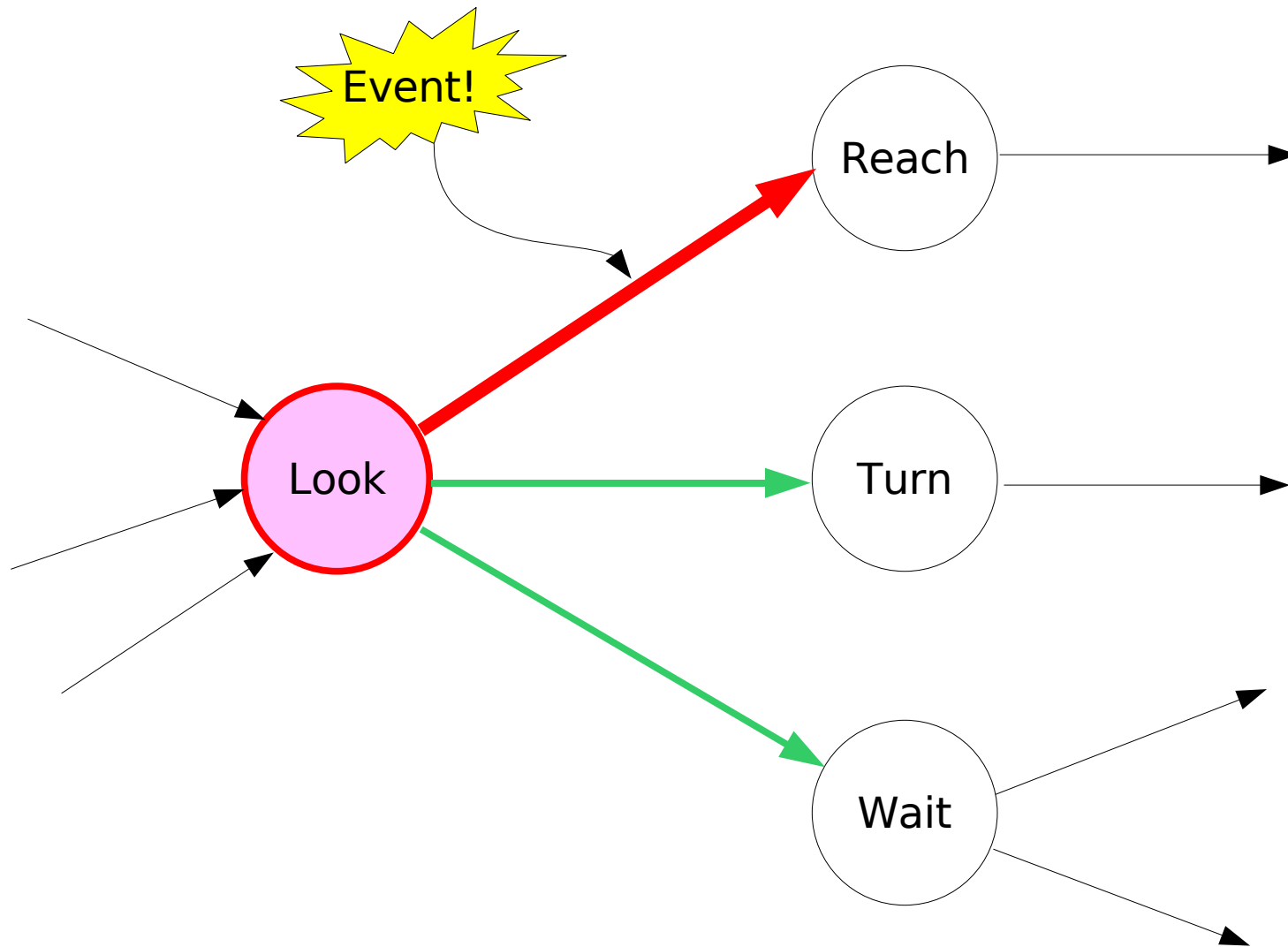
Random things happen....



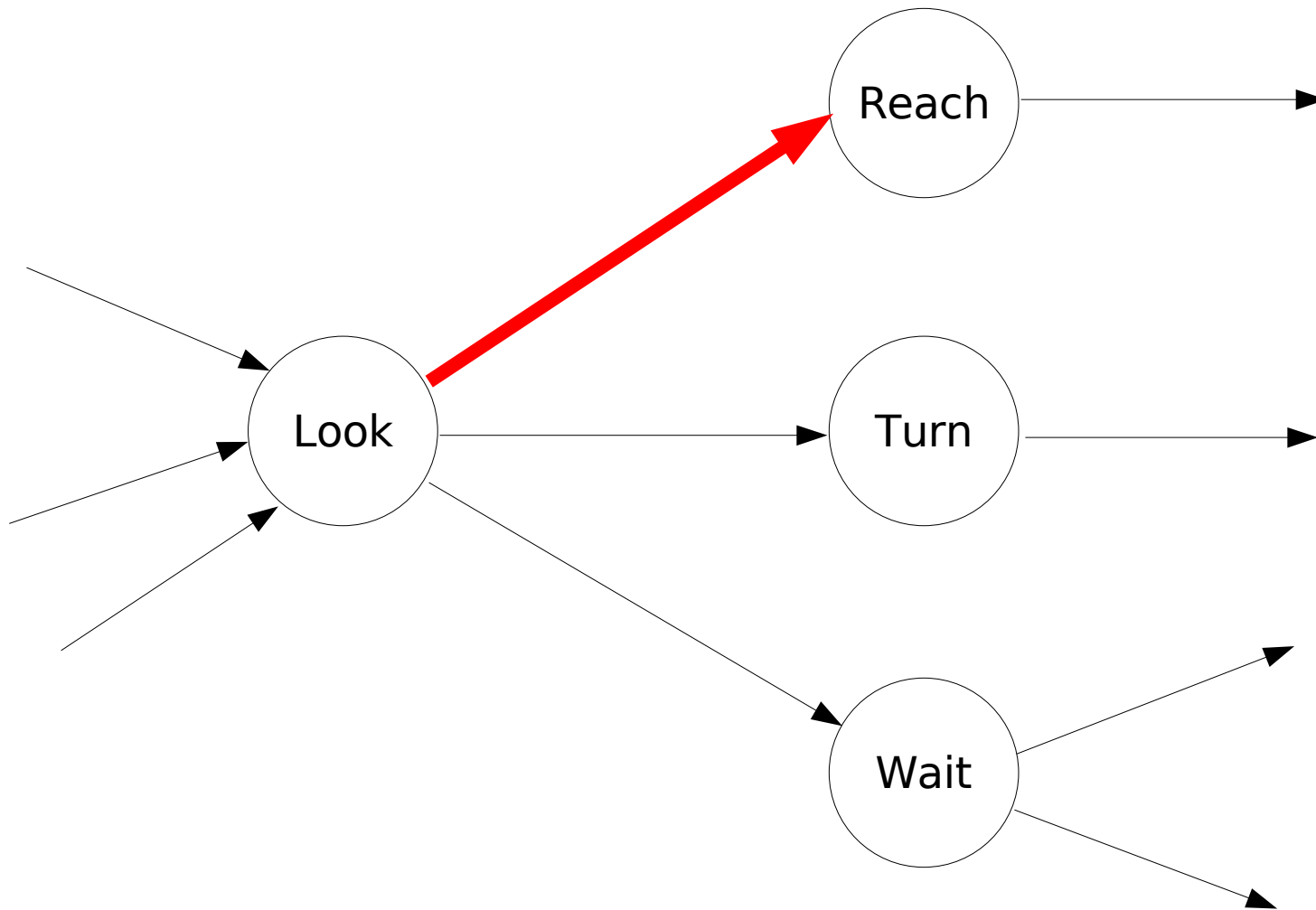
And then, something we've been looking for...



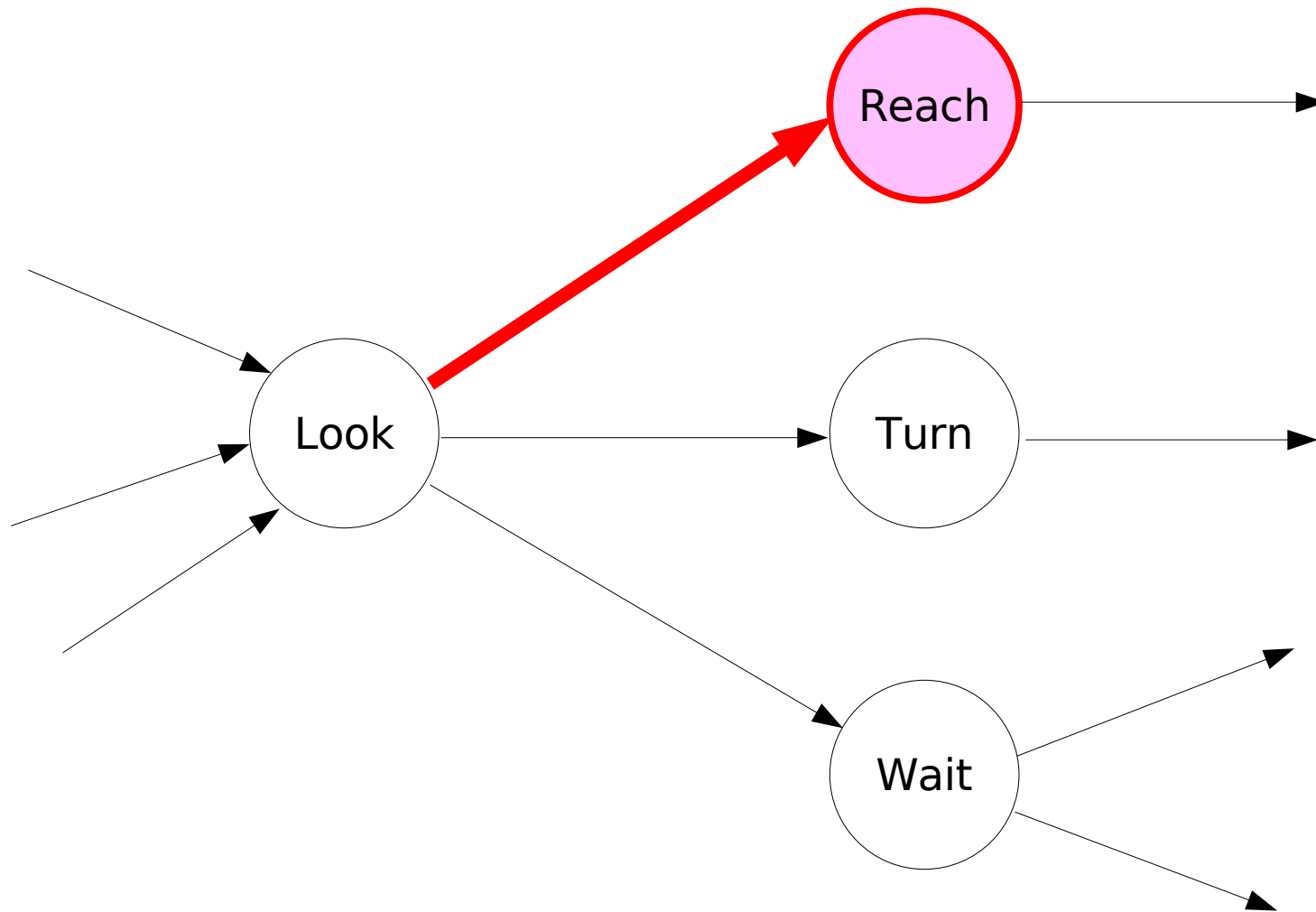
Transition decides to fire.



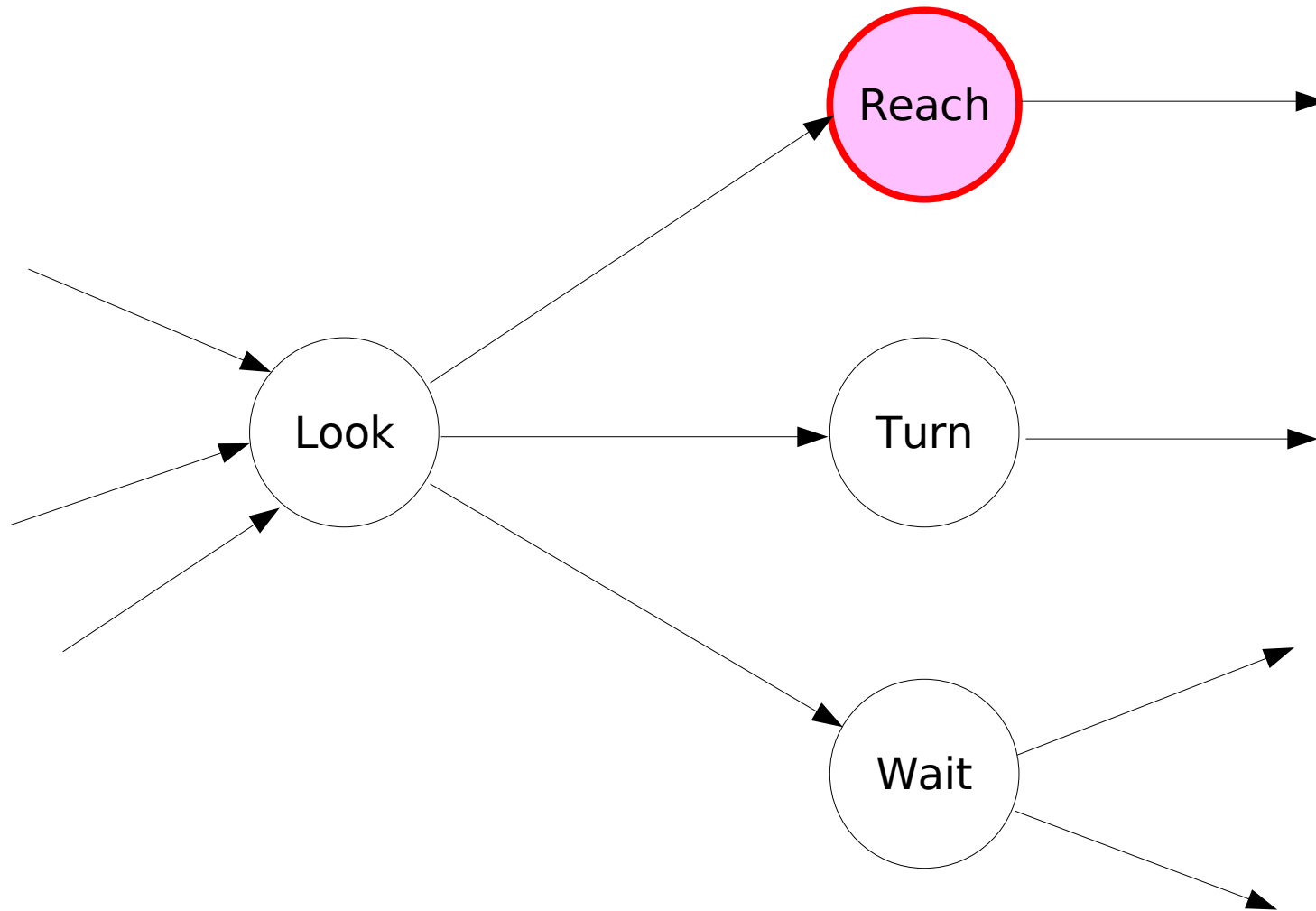
Transition deactivates the source node, Look.



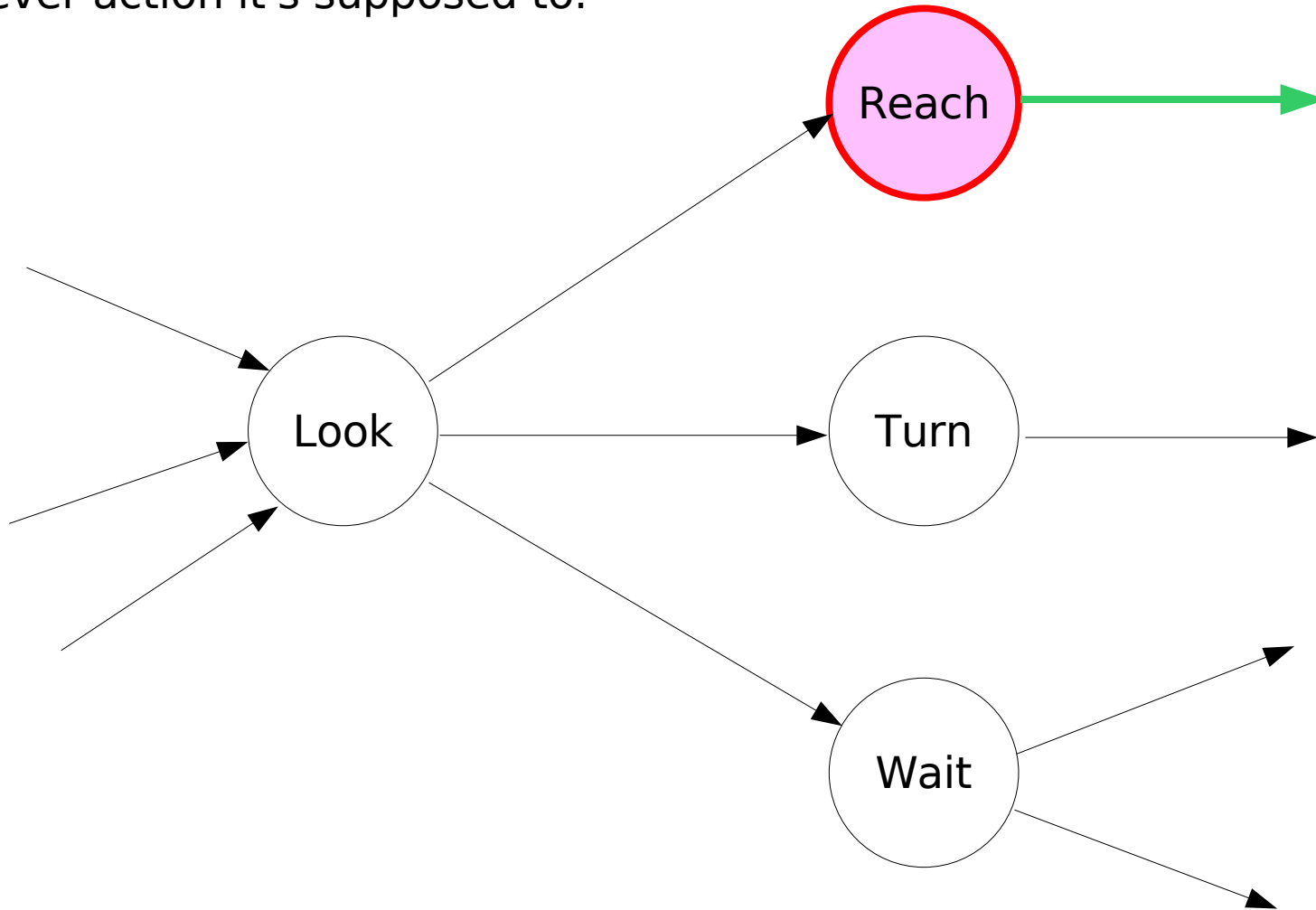
Transition activates the destination node, Reach.



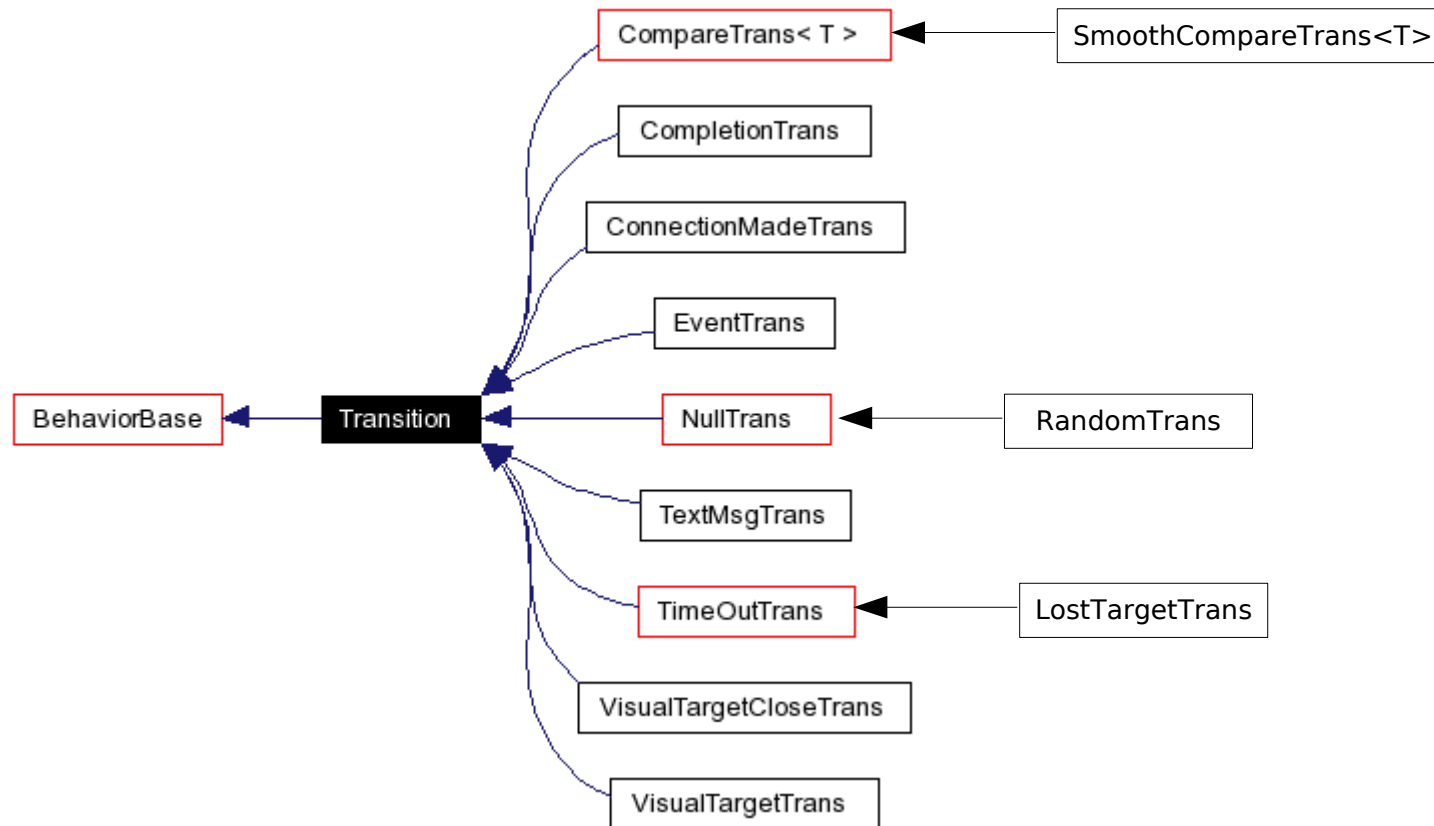
Transition deactivates.



Reach activates its outgoing transition, which starts listening for events as Reach performs whatever action it's supposed to.



Transition Types



State Machine Compiler

- Tekkotsu programmers don't normally write C++ code to build state machines one node or link at a time.
- They use a shorthand notation instead.
- The shorthand is turned into C++ by a state machine compiler.
- But to understand what the shorthand is doing, we need to build our first state machine by hand.



Programs As State Machines

Your program is the parent StateNode:

```
#include "Behaviors/StateMachine.h"

class BarkHowlBlinkBehavior : public StateNode {

public:
    BarkHowlBlinkBehavior() :
        StateNode("BarkHowlBlinkBehavior") {}
}
```

Setup and Teardown

- Programs must include a `setup()` function to construct the state machine as a child of the parent state node.
- `setup()` is called automatically the first time the parent's `start()` is called.
- A `teardown()` function is automatically provided to destroy the state machine. Called by `~StateNode()`.

Registering Nodes and Links

- Each node created by setup() must be registered with its parent using the addNode() method.

```
SoundNode *bark_node = new SoundNode("bark", "barkmed.wav");  
addNode(bark_node);
```

- Transitions are registered with their source nodes via the source node's addTransition() method.

```
bark_node->addTransition(new TimeoutTrans(howl_node, 5000));
```

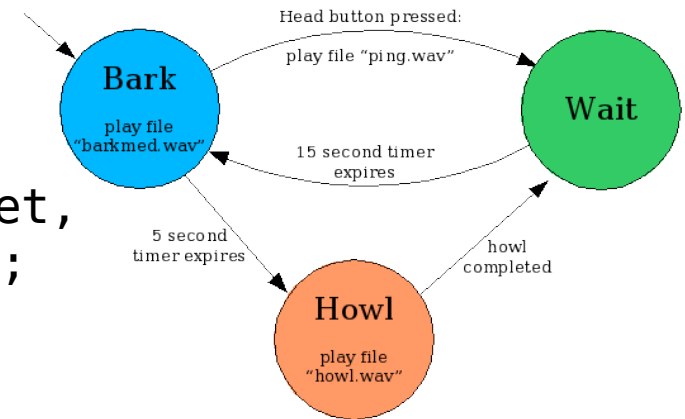
- The variable startnode must be set to point to the starting node of the state machine.

Setup Example

```
virtual void setup() {
```

```
    SoundNode *bark_node = new SoundNode("bark", "barkmed.wav");  
    SoundNode *howl_node = new SoundNode("howl", "howl.wav");  
    StateNode *wait_node = new StateNode("wait");  
    addNode(bark_node); addNode(howl_node); addNode(wait_node);
```

```
    EventTrans *btrans =  
        new EventTrans(wait_node,  
                        EventBus::buttonEGID,  
                        ChiaraInfo::GreenButOffset,  
                        EventBus::activateETID);  
    btrans->setSound("ping.wav");  
    bark_node->addTransition(btrans);
```



```
    howl_node->addTransition(new CompletionTrans(wait_node));  
    wait_node->addTransition(new TimeoutTrans(bark_node, 15000));
```

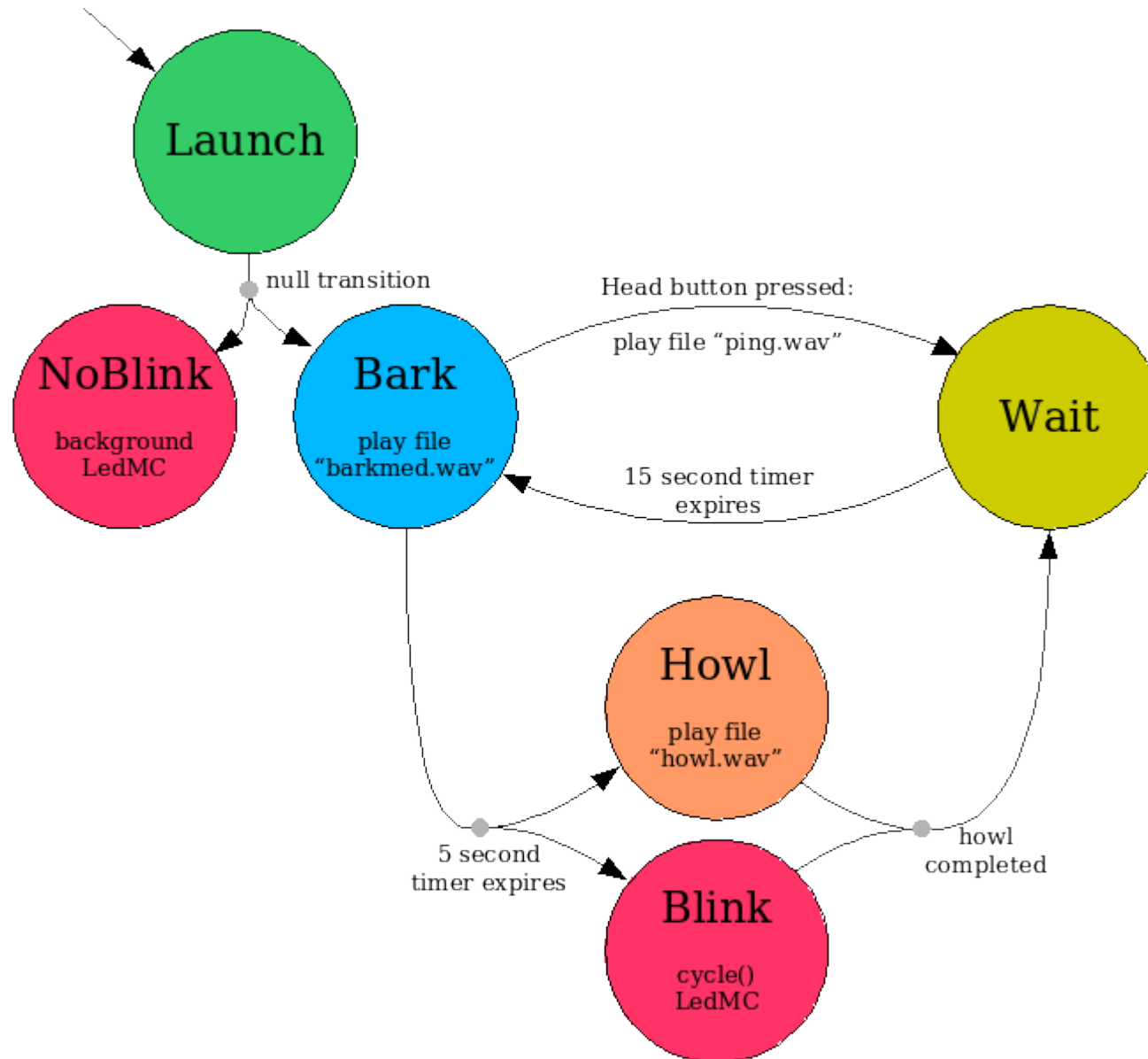
```
    startnode = bark_node;
```

```
}
```

Extensions to the Basic Formalism

- Extension 1: multi-states (parallelism).
 - Several states can be active at once.
 - Provides for parallel processing (but coroutines, not threads).
- Extension 2: hierarchical structure.
 - State machines can nest inside other state machines.
- Extension 3: message passing.
 - When a state posts an event that triggers a transition, it can include a message that will be passed to the destination state.
 - This makes state transitions resemble procedure calls.

Multi-State Machines



Blink Using LedEngine::cycle()

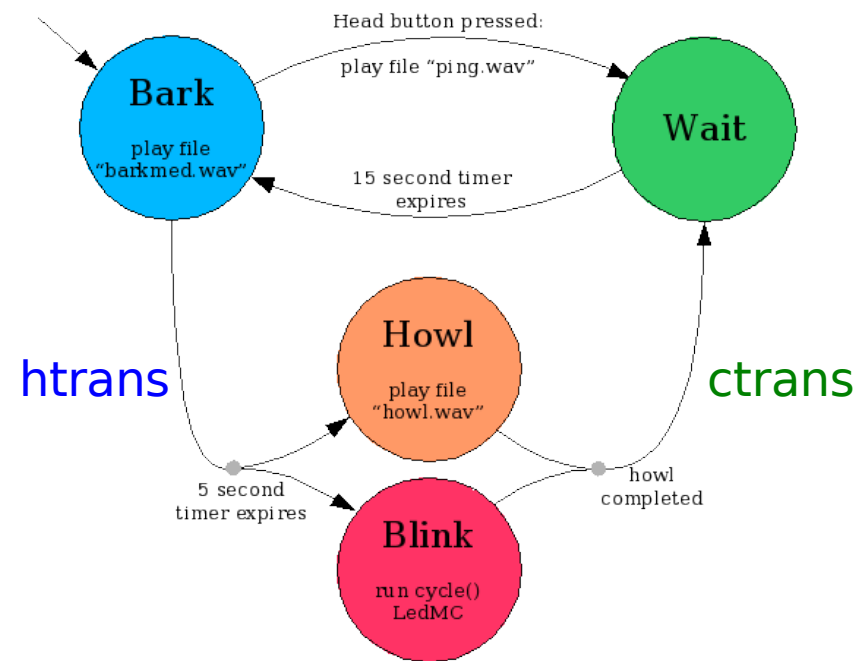
- Blink uses a motion command called LedMC, which is a child of LedEngine.
- The LedEngine::cycle() method never completes.
- When the howl completes, we want to leave both the howl state and the blink state.
- We can do this by telling CompletionTrans that only one of its source nodes needs to signal a completion in order for the transition to fire.
- When it does fire, it will deactivate both source nodes.

Setting Up the Blink

```
LedNode *blink_node = new LedNode("blink");  
addNode(blink_node);  
blink_node->getMC()->cycle(RobotInfo::AllLEDMask,1500,1.0);
```

```
TimeoutTrans *htrans = new TimeoutTrans(howl_node,5000);  
htrans->addDestination(blink_node);  
bark_node->addTransition(htrans);
```

```
CompletionTrans *ctrans = new CompletionTrans(wait_node,1);  
howl_node->addTransition(ctrans);  
blink_node->addTransition(ctrans);
```



Cleaning Up the Blink: Turn The LEDs Off

```
LedNode *noblink = new LedNode("noblink");
```

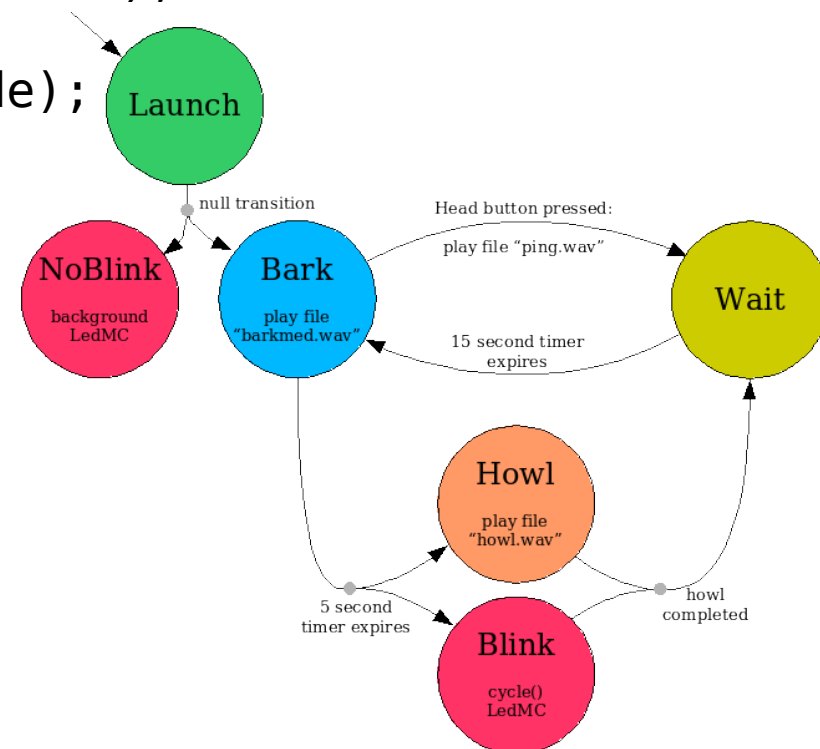
```
noblink->getMC()->set(RobotInfo::AllLEDMask, 0.0);  
noblink->setPriority(MotionManager::kBackgroundPriority);
```

```
StateNode *launcher = new StateNode("launcher");
```

```
NullTrans *ntrans = new NullTrans(bark_node);  
ntrans->addDestination(noblink);
```

```
launcher->addTransition(ntrans);
```

```
startnode = launcher;
```



Shorthand Notation

bark: SoundNode(\$,"barkmed.wav")

howl: SoundNode(\$,"howl.wav")

wait: StateNode

bark =T(5000)=> howl

bark =B(GreenButOffset, activateETID)=> wait

Shorthand Notation

- Node definition:

nodename: NodeClass(constructor_args)[initializers]

- Transition, short form examples:

source =C=> target

source =T(n)=> target

source =E(g,s,t)=> target

- Transition, long form:

source >== transname:

TransitionClass(constructor_args)[initializers] ==> targetnode

- Multiple sources/targets:

source >==Transition==> {targ1name, targ2name, ...}

\$ and \$\$

- Use $\$$ to refer to the name of the current node, e.g., these are equivalent:

foo: Statenode

foo: StateNode($\$$)

foo: StateNode("foo")

bar: SoundNode($\$$, "howl.wav")

bar: SoundNode("bar", "howl.wav")

Must be present
to allow second
argument

- In long form, use $\$\$$ to refer to the destination node of a transition, e.g., these are equivalent:

foo >==EventTrans($\$\$$,EventBase::buttonEGID)==> bar

foo >==EventTrans(bar,EventBase::buttonEGID)==> bar

More Shorthand

<code>>==NullTrans==></code>	<code>=N=></code>
<code>>==CompletionTrans==></code>	<code>=C=></code>
<code>>==CompletionTrans(\$,\$\$,n)==></code>	<code>=C(n)==></code>
<code>>==TimeoutTrans(\$,\$\$,t)==></code>	<code>=T(t)==></code>
<code>>==EventTrans(\$,\$\$,g,s,t)==></code>	<code>=E(g,s,t)==></code>
<code>>== EventTrans(\$,\$\$, EventBase::buttonEGID,s,t) ==></code>	<code>=B(s,t)==></code>
<code>>== TextMsgTrans(\$,\$\$,str)==></code>	<code>=TM(str)==></code>
<code>>==RandomTrans==></code>	<code>=RND=></code>
<code>>==SignalTrans<T>(\$,\$\$) ==></code>	<code>=S<T>=></code>
<code>>==SignalTrans<T>(\$,\$\$,v)==></code>	<code>=S<T>(v)==></code>

```
virtual void setup() {  
#statemachine  
launcher:StateNode =N=> {noblink, bark}  
  
noblink: LedNode [setPriority(MotionManager::kBackgroundPriority);  
                  getMC()->set(RobotInfo::FaceLEDMask,0.0);]  
  
bark: SoundNode($,"barkmed.wav")  
      =B(GreenButOffset,activateETID)[setSound("ping.wav");]=> wait  
  
wait: StateNode =T(15000)=> bark  
  
bark =T(5000)=> {howl, blink}  
  
howl: SoundNode($,"howl.wav")  
  
blink: LedNode [getMC()->cycle(RobotInfo::AllLEDMask, 1500, 1.0);]  
  
{howl, blink} =C(1)=> wait  
#endstatemachine  
  
startnode = launcher; } // end of setup()
```


Compiling Your FSM

- The Makefile looks for files with names of form *.fsm and automatically runs them through the state machine compiler, called “stateparser”.
- BarkHowIBlinkBehavior.h.fsm generates a pure C++ file called BarkHowIBlinkBehavior.h.
- The .h file is stored in:
 ~/project/build/PLATFORM_LOCAL/TARGET_CHIARA
- You can run the stateparser directly:

```
Tekkotsu/tools/stateparser BarkHowIBlinkBehavior.h.fsm -
```

Other Transition Types

- NullTrans fires immediately.
 - Useful for nodes that just initiate an action and then move on.
- RandomTrans enters one of its target states at random.
- CompareTrans compares a memory location with a value, and fires if the specified test is met. For example, to transition when IR indicates 200 mm from an obstacle:

```
CompareTrans(node37,  
             &state->sensors[FrontIRDistOffset],  
             CompareTrans::LT,  
             200,  
             EventBase(EventBase::sensorEGID,  
                       SensorSourceID::UpdatedSID,  
                       EventBase::statusETID))
```

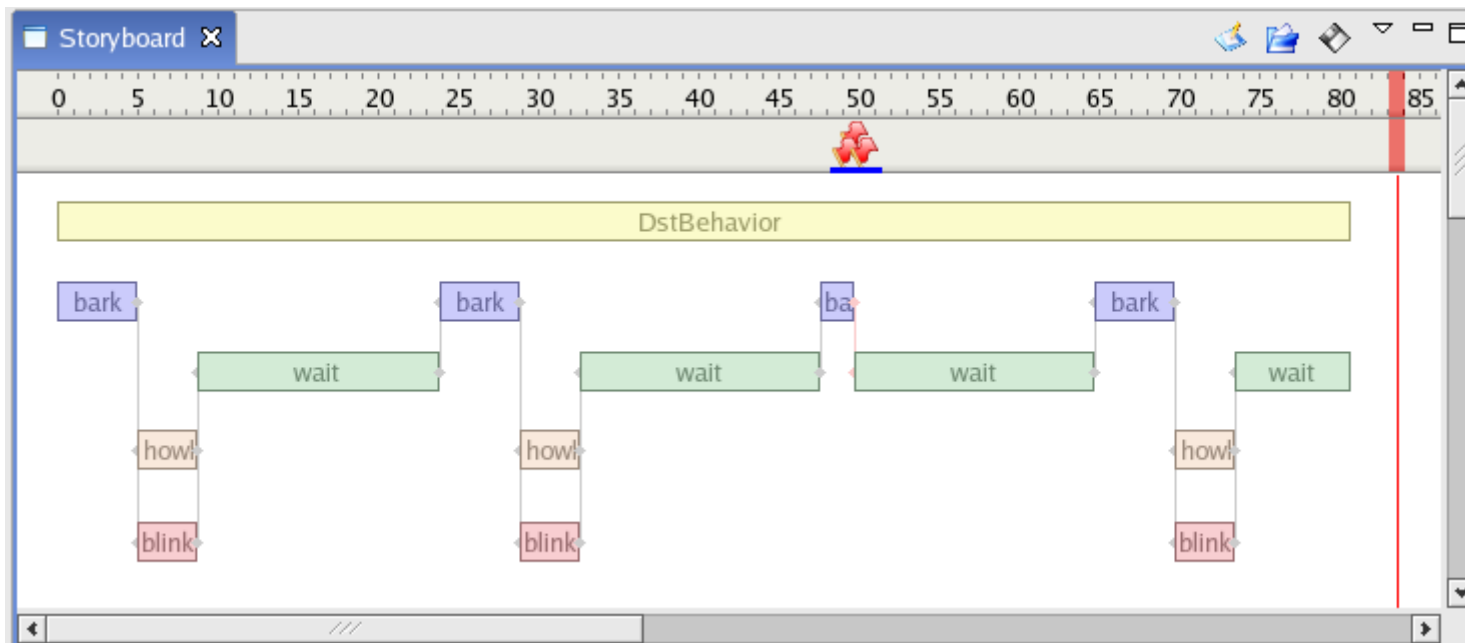
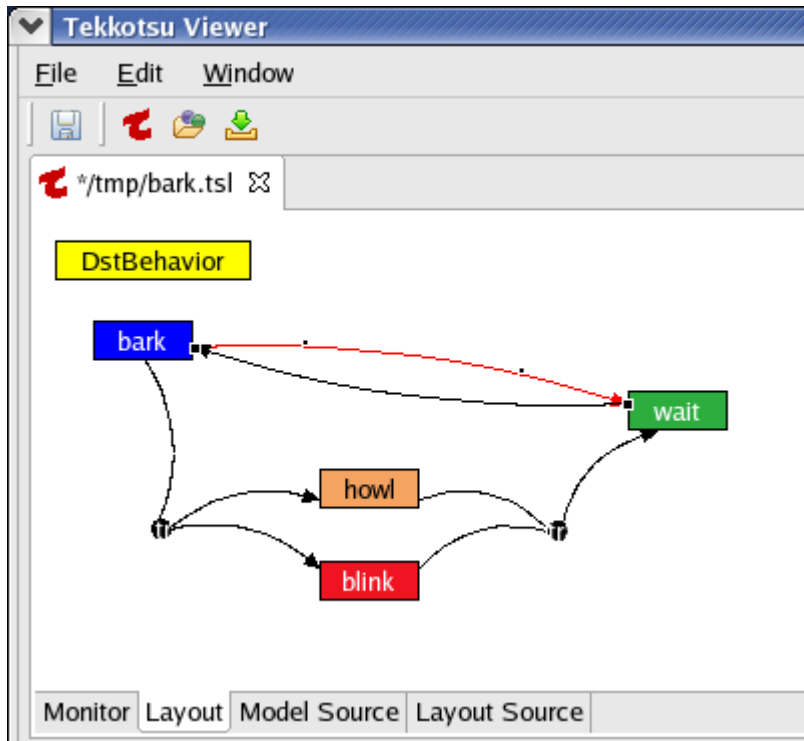
- SignalTrans looks for a specified DataEvent
 - Useful for implementing “switch” statements

State Machine Events

- Entering or leaving a state generates a stateMachineEGID event.
 - activateETID for entering
 - deactivateETID for leaving
- Firing of a transition generates a stateTransitionEGID event.
- SignalTrans looks for a stateSignalEGID event
- You can use the Tekkotsu Event Logger to monitor these events:

Root Control > Status Reports > Event Logger

Storyboard Tool: State Machine Layout



Storyboard Tool: Storyboard Display

The screenshot displays the Tekkotsu Viewer interface, which is used for visualizing and debugging state machines. The main window is titled "Tekkotsu Viewer" and contains a menu bar (File, Edit, Window) and a toolbar with icons for saving, undo, redo, and opening files. The central area shows a state machine model with nodes such as Pink, Follow, Sit, Funny, Timer, Sound, Up, Down, Sniff, Look, and Punch, connected by directed edges. To the right of the model is a control panel with fields for Host (localhost) and Port (10080), a Name field (Explore State Machine), and buttons for Download Model, New Trace, and a refresh icon. Below the model are tabs for Monitor, Layout, Model Source, and Layout Source. On the right side, there is a Properties panel with a Runtime View tab, showing details for the current selection (timer at 14.256s), including activate/deactivate times and types for Timer, Timer->Sit, and Sit. At the bottom, there is a Storyboard panel with a timeline from 0 to 60 seconds. A red vertical line indicates the current time, which is approximately 27.5 seconds. The storyboard shows a sequence of state transitions and activations, with a green bar representing the Timer state and a blue bar representing the Sit state. To the right of the storyboard is an Image Preview panel, which is currently empty.

Storyboard Tool: Snapshots

The screenshot displays the Tekkotsu Viewer application interface, which is used for developing and running state machines. The interface is divided into several panels:

- Top Panel:** Contains the menu bar (File, Edit, Window) and a toolbar with icons for file operations. Below this is a browser-like address bar showing the file path: `*/afs/cs.cmu.edu/user/dst/S...`.
- Host/Port Section:** Shows the host as `localhost` and the port as `10080`. The name of the state machine is `Explore State Machine`. There are buttons for `Download Model`, `New Trace`, and a refresh icon.
- Storyboard View (Bottom):** A timeline view showing the execution of the state machine. A yellow bar at the top represents the `Logging Test` state, which is active from time 0 to 57.206s. Below this, a sequence of states is shown: `Waiting` (green), `Image` (blue), `Webcam` (red), `Message` (purple), `Message` (purple), and `Image` (blue). Each state is connected to a corresponding state in the top view by arrows.
- Properties Panel (Right):** Shows the current selection at `:9.491s`. It lists the following properties:
 - `Image:Image`: record at: 8.457s, type: image
 - `Waiting`: activate at: 8.495000000000000, deactivate at: 18.201s, type: state
 - `Logging Test`: activate at: 0.0s, deactivate at: 57.206s
- Image Preview (Bottom Right):** A small window showing a live video feed from a webcam, displaying a room with a computer monitor and other equipment.